Embedded-system developers looking for a way to add graphics to their systems have a range of methods to choose from; no absolute prescription for success exists. The traditional C-based graphical-user-interface (GUI) tool kit and associated development tools are strong contenders. Java is gaining acceptance, and smaller memory requirements make it a serious player. These two paradigms present attractive options for compelling solutions of next-generation embedded systems.

An HTML approach breaks away from these programmatic paradigms and builds on the huge presence HTML has in the World Wide Web. This viable approach solves many of the problems of traditional approaches and has the benefit of clearly separating the user interface from the application. Traditional real-time operating systems (RTOSs), designed for mission-critical applications, are powerful tools for building the devices becoming prevalent today. Combining an RTOS with HTML allows you to add powerful, appealing, and visually distinctive GUIs to embedded devices without compromising reliability or performance.

**Fig. 1**

An HTML user interface uses JavaScript and LiveConnect to communicate with other languages by effectively exposing each language’s objects to the other language.

**TABLE**

<table>
<thead>
<tr>
<th>HTML USER INTERFACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMBEDDED REAL-TIME APPLICATION</td>
</tr>
<tr>
<td>LIVECONNECT</td>
</tr>
<tr>
<td>VxWORKs</td>
</tr>
</tbody>
</table>
video-disk players with “true GUI” set-up menus—to name a few—are getting smarter. Silicon and display prices are falling. New display technologies, such as light-emitting polymers, may make it possible to have a display on just about any device, even without flat space to attach the display to.

The traditional embedded markets are also changing. Printers, copiers, and test instruments have traditionally had few or no local GUIs. Copiers have relied on a simple LCD panel, and a workstation often remotely manages networked devices. As display prices fall, these systems are acquiring increasingly sophisticated local user interfaces, either as alternatives to remote management or as facilitators of initial setup. High-end imaging devices offer high-resolution displays with GUIs that feature tabbed boxes, buttons, menus, and pop-up dialogs. This activity is all happening in the context of strong growth in the smart-embedded-system market—systems with 32-bit CPUs and firmware with a distinct homegrown or off-the-shelf OS layer.

**CONSIDER YOUR GUI CHOICES**

As embedded products evolve, developers need to choose a paradigm to use for embedded-GUI development. Historically, designers have based GUIs on C or C++ application-programming interfaces (APIs), which is the case with the X Windows System (available on all Unix systems), Microsoft Windows (Microsoft Corp, www.microsoft.com), and various portable GUI libraries. A few interfaces use more advanced languages, such as Smalltalk and TCL/Tk. In the embedded-system world, however, such standardization is rare. The market is younger, and standardization with the implicit lock to a large application base is not the issue it is in the desktop world.

As a result, a number of distinct but co-existing technologies may serve embedded-systems graphics development. You can place these technologies into three camps. The first camp, traditional 3GL methods based on C and C++, will continue to have a strong presence for compelling reasons, including high performance and reliability. Java represents a second new-application-environment camp, for which a completely new programming environment provides a secure base for GUI development. And finally, new GUI paradigms that break away from the programmatic approach to GUI development are beginning to emerge.

**THE THIRD GENERATION**

Traditional 3GL methods are still going strong. X Windows has gone embedded; Windows CE offers a subset of Win32; and a range of proprietary approaches exist from RTOS vendors, such as Photon (QNIX Software Systems Ltd, www.qnx.com), Maui Microware Systems Corp, www.microware.com), and Poptic (Integrated Systems Inc, www.isi.com). Zinc Software Inc (www.zinc.com), which natively supports all Windows platforms, Unix X/Motif, DOS, and several embedded OSs, provides a complete object-oriented C++ API.

Many questions arise when you use this traditional paradigm for embedded systems, however. For example, which API should you adopt? The proprietary approaches offer similar functionality to each other. Several have advanced GUI-building solutions, but their proprietary nature may be a problem. You can solve this problem, however, with a GUI that is available across platforms.

X Windows is an open standard, and XLib/Xt offers no GUI features. Adding Motif (Open Software Foundation, www.opengroup.org) as the GUI component leads to considerable bloat. Motif is appropriate on the desktop, but when the pushbutton widget alone has 65 configurable resources (XmPushButton main page), Motif takes too much memory space.

Windows CE’s Win32 is a “proprietary standard” that has huge market dominance in the desktop world but may be unsuitable for embedded systems. Microsoft has made much hoopla of your ability to leverage Win32 code for use in embedded devices. Under close inspection, however, this argument is weak. Even ignoring the fact that the Windows CE API is only a subset of Win32, you would hardly run Office 98 in a palmtop with 4 Mbytes of RAM and a 160 × 160-pixel LCD. Embedded applications, even those as ubiquitous as word processing, are usually custom-designed for their environment. By adopting a more universal overall API, the embedded community could resolve the challenges associated with a proprietary API.

Zinc (Zinc Software Inc) is a user-interface class library and a visual-development tool. Written in C++, Zinc provides serious developers with an easy-to-use, object-oriented API that can generally replace more difficult native APIs. Zinc is a complete GUI whose unique strengths lie in its ease of use, portability, and internationalization. Zinc comprises GUI libraries, a visual design tool, portable make utility, hypertext-based online documentation, and numerous examples and tutorials. You can easily scale and configure Zinc to meet the exact GUI requirements of an application. Designed for memory-constrained environments, Zinc can fit a configuration comprising commercial OS and Zinc libraries into less than 1 Mbyte of memory.

Zinc also makes it possible to create sophisticated GUIs using a range of user-interface objects—from windows and buttons to notebooks and tables. You can customize any user-interface object to create a unique appearance and feel or to emulate any popular desktop style.

**POINTER AND CORE DUMPS**

Unfortunately, the traditional C API approach sacrifices a degree of reliability for high performance and a small memory footprint. Embedded and real-time devices often need much higher reliabil-
Graphics in embedded systems

The new-application-environments camp currently has only one major member: Java.

Move to New Environments

The new-application-environments camp currently has only one major member: Java. This method involves a secure programming paradigm (no pointers, so you can’t corrupt the memory model) and a “holistic” approach to the application environment, encompassing graphics and any other APIs an application is likely to need.

JavaSoft, a division of Sun Microsystems (www.javasoft.com), has been promoting Java’s embedded-system suitability, and new versions of Java are much smaller. Java now appears as the GUI in some embedded devices, such as the Al tura Lottery Terminal from Gtech Corp (www.gtech.com), which will begin shipping this year. As Java’s Abstract Windowing Toolkit (pAWT) becomes more portable, Java suppliers can use less-memory-intensive underlying graphics-software stacks. Furthermore, Java’s memory footprint will likely become even smaller, making Java a serious contender for embedded-systems graphics development.

Considering these changes and falling hardware costs, Java is set to go where its developers intended from the start—the embedded-systems environment. Java will never be the right answer for everyone because of the trade-offs of programming in each application environment. Java is still a relatively “fiddly” way to build a GUI unless you use a GUI-builder tool. One of the big advantages of Java is its standardized API. Also, tools and applications, such as GUI builders, are becoming increasingly available. Although Java’s portability is an asset in many environments, such as for online software updates, Java is not relevant to every application. Overall, room exists for a third approach to embedded GUI development.

Thanks to the Internet, HTML is the most popular GUI-development language (Figure 1). A page on the Web is a GUI in itself. An increasing number of Web sites are exploring the capabilities of HTML for constructing interactive GUIs. Any number of airline booking systems and online shops use the basic form facilities of HTML, which often include some JavaScript enhancements.

Some sites achieve a more sophisticated style by using dynamic images; Anders Viquer’s site at http://aqa.dse offers demos of several GUI-style “control panels” to illustrate the look-and-feel possibilities. These demos use images to implement artistic buttons that link to JavaScript code. When the mouse moves over the buttons, the code provides live feedback by flipping the button image and popping up an explanatory image in a central window.

The first objection to this approach is that although such pages are attractive, they do nothing besides changing some images on the screen or submitting some information to a remote server. Apart from a few slick exceptions, this drawback is true of most of the pages on the Web. But the power of this approach becomes apparent when you link HTML and JavaScript, running on an embedded device, to some underlying application code. You can provide the link using LiveConnect or some similar technology.

LiveConnect allows JavaScript to communicate with other languages, such as Java or C/C++, by effectively exposing each language’s objects to the other language. Because an HTML object can have an attachment of arbitrary JavaScript, a mouse click over one of those active buttons could trigger a call to underlying application code to start a motor, for example. The application could likewise invoke JavaScript code to change an on-screen image or go to a new HTML page, maybe within a frame, for example.

Consider the Benefits of HTML

HTML is arguably the most open standard. There are approximately 55 Web browsers currently available for desktop systems (http://browserwatch.internet.com/), plus browsers for PDAs such as the Pilot and the “embedded” Web browsers in devices such as Web TV and its competitors. A range of authoring tools is available, and the amount of content written in HTML is growing. In short, creating a GUI in HTML is relatively easy; plenty of tools and people with the right skills are available.

HTML also neatly solves the reliability issue. In an embedded device, such as a high-end copier that is about to acquire a high-resolution local display, the application code most likely exists and is probably in C. In-house real-time engineers who understand such issues as interrupt handling and memory management develop and enhance this code.

When a GUI team designs the HTML/JavaScript front end, real-time designers simply need to add a LiveConnect binding to their code, which presents the GUI
team with a secure “sandpit”—an isolated design environment—in which to do its work. That is, the GUI team can access all the functions that the LiveConnect binding declares but nothing else. And, because HTML and JavaScript have no pointers, GUI builders can’t accidentally corrupt the memory model and crash the system. They can instead concentrate on getting the correct look and feel, while the real-time people concentrate on the application enhancements.

This narrow link between applications and GUIs has another benefit. With the traditional C-based approach, the GUI compiles and links to the application code, which forms one monolithic executable image. Using HTML and JavaScript, you can effectively store the GUI as a collection of files in a device’s file system, probably in flash memory, thus easing GUI updates.

CONTROL THE LOOK AND FEEL

This approach results in a look and feel that differs widely from that of the normal desktop environment. Animated artistic buttons and flipping images and frames, among other features, replace the menus, scroll bars, rectangular push buttons, and task bars of traditional GUI approaches. For the embedded market, this change might turn out well.

For example, when you go to use a gas pump, you don’t expect a task bar, application windows, or icons. You want big, obvious buttons that tell you what to do. The newest gas pumps offer this type of interface on full-color CRT displays with options for ordering a car wash or goods from the shop, all available through simple on-screen menus with pictures of what’s for sale. HTML excels in this type of interface; HTML allows you to easily use images to build an interface with a theme tailored to an application or to use images that use corporate branding.

Controlling the look and feel is perhaps the biggest issue. An approach based on a traditional GUI platform is constrained by the style of that platform. For example, using Windows CE makes it practically mandatory to stick to the Windows look and feel. HTML offers total flexibility in how you can construct a user interface and in the styles and
themes you use for the interface. This flexibility allows a vendor to achieve true market branding and product differentiation.

CAN HTML DELIVER?

HTML is more limited than traditional GUI tool kits; it has no moving or pop-up elements, such as menus and dialogues. However, a surprising number of features are possible using the various techniques available. You can implement a tabbed subwindow, for example, by using composite images or frames. Given the application-specific nature of most embedded systems, HTML is more than adequate.

You use JavaScript to achieve all of the dynamic aspects of HTML, including the link between the HTML presentation and the underlying application code. You can achieve similar dynamic aspects by providing a C API to the HTML layer, but you lose the benefits of standardization and "pointer-proofness." Using a C API also splits the behavioral parts of the GUI from the presentation and links them to the application code.

Looking ahead, Dynamic HTML, when it becomes available, will provide more flexibility by adding the ability to manipulate the screen more freely. Layers will allow you to implement moving or overlapping panels, such as dialogues and menus, or to replace some of what you would currently do using frames. The Document Object Model, which allows you to restyle or rewrite sections of HTML, is likely to have less of an impact because these techniques are less relevant to embedded-style user interfaces.

Author’s biography
Ian Smith is senior software engineer at Wind River Systems (Alameda, CA, www.wrs.com), where he has worked for one year as project leader for the company’s HTML browser for VXWorks. He has a BSc in computers and cybernetics from the University of Kent (Canterbury, UK).